THESIS

NEURAL NETWORKS FOR MALWARE DETECTION USING STATIC ANALYSIS

by

Pawel Kalinowski

March 2019

Thesis Advisor:	Neil C. Rowe
Co-Advisor:	Christopher S. Eagle

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2019	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE NEURAL NETWORKS FOR MALWARE DETECTION USING STATIC ANALYSIS			**5. FUNDING NUMBERS**
6. AUTHOR(S) Pawel Kalinowski			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Malware is software that enables adversaries to execute their goals by affecting their target devices' confidentiality, integrity, or availability. Malware is constantly evolving and detection methods must find ways to detect the new variants. This research developed a new method of detecting malware using a neural-network architecture. The method is not signature-based, unlike most existing methods, and would aid in finding previously unseen malware. It analyzes software using three separate static-analysis methods to obtain a list of features, which when input into the neural network are used to classify the software as malware or not malware. The three methods were the binary-to-grayscale, statistical-N-grams, and dynamic-link-libraries. The binary-to-grayscale approach performed poorly. The other two strategies performed better, but had room for improvement; statistical-N-grams and dynamic-link-libraries showed complementary results that suggest combining them would yield a more effective detection method.

14. SUBJECT TERMS malware, convolutional neural networks, static analysis, detection, classification			**15. NUMBER OF PAGES** 67
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

NEURAL NETWORKS FOR MALWARE DETECTION USING STATIC ANALYSIS

Pawel Kalinowski
Civilian, Department of the Navy
BSE, Boston University, 2012
BSE, Wayne State University, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2019

Approved by: Neil C. Rowe
Advisor

Christopher S. Eagle
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Malware is software that enables adversaries to execute their goals by affecting their target devices' confidentiality, integrity, or availability. Malware is constantly evolving and detection methods must find ways to detect the new variants. This research developed a new method of detecting malware using a neural-network architecture. The method is not signature-based, unlike most existing methods, and would aid in finding previously unseen malware. It analyzes software using three separate static-analysis methods to obtain a list of features, which when input into the neural network are used to classify the software as malware or not malware. The three methods were the binary-to-grayscale, statistical-N-grams, and dynamic-link-libraries. The binary-to-grayscale approach performed poorly. The other two strategies performed better, but had room for improvement; statistical-N-grams and dynamic-link-libraries showed complementary results that suggest combining them would yield a more effective detection method.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Malware is software that intends to act in a harmful or intrusive manner toward a target computer device. Categories of malware are Trojan horses, viruses, worms, ransomware, spyware, adware, rootkits, keyloggers, and backdoors. Malware enables adversaries to execute their goals against a target, allowing them to compromise the device's computer-security triad of confidentiality, integrity, and availability.

Computer scientists over the years have developed detection techniques to identify malware executables (binaries). Traditionally, antivirus software relies on signatures to identify malware. A signature is a unique pattern of bits that can identify a file, like a fingerprint. However, like the flu virus, malware may be designed to mutate to help it survive and fool a system's detection mechanisms while retaining its functionality. A few minor modifications to malware can change its signatures and that often suffices to cause a signature-based detection system to fail.

As computation speed has increased and machine-learning algorithms have gained popularity, antivirus detection has evolved in its scope to attempt detection of malware previously unidentified (zero days) using signatures. To aid in this identification, various machine-learning algorithms are available. These algorithms have gained wide use in fields such as marketing, finance, genetics, and manufacturing. Data is analyzed to obtain a set of features that are then used as input to predict a result. Usually, the algorithms attempt to duplicate a specified set of outputs for a set of inputs given in a training set algorithms like support-vector machines, nearest-neighbor inference, and Naïve Bayes have been popular (Fatima & Pasha, 2017; Jahan, 2018, Muja & Lowe, 2014). Lately, the use of neural networks has begun to show promise in a variety of applications.

Neural networks are a form of neural-network learning methods where a network is trained on data that have already been classified by a reliable method (has a "ground truth") (Kelleher, Namee, & Darcy, 2015). This study focused on malware identification, and the ground truth was the identification of whether a binary image of an executable file was malware or not. The training of supervised models relies on the adjustment of weights. A set of input values and a set of weights on them, in multiple layers, are used to calculate output values. For training a neural network model with malware and benign binaries, a gradient-descent optimization algorithm can modify the weights within the “hidden” and “output” layers in the neural network. This adjustment of weights creates a non-linear function on the inputs that can extrapolate to new but similar kinds of malware. Many types of neural network architectures have evolved over time.

This thesis did three static analyses that extracted features from executable binaries and used them as input to three neural network models in an attempt to identify malware. Binaries were restricted to Windows 7 executables. One model used the Cuckoo Sandbox software to extract API library calls, another used basic statistical analysis of N-grams in the binary, and one used a gray-scale image constructed from the binary. The models used feed-forward neural network architectures, and the image model also used a convolutional neural network.

This thesis first discusses the background and the work related to malware detection in recent years. Then, the proposed methodology for the study is presented. Finally, the results and analysis are provided, along with the code used to obtain the results. The binary-to-grayscale image strategy performed the poorest of the three. Although the accuracies are not extraordinary, the other two strategies performed similarly to each other. The statistical-N-gram and dynamic-link-library strategies may even have complementary qualities, which may improve performance in a hybrid neural-network architecture.

II. BACKGROUND

Malware is computer software that performs malicious actions on a victim's computer device without their consent. The actions performed depend on the type of malware. The types include ransomware, downloader, bot, dropper, worm, keylogger, or adware. Ransomware encrypts a victim's data and holds the decryption keys for ransom. A downloader looks benign, but it will download something more malicious later. A "bot" allows malicious remote control of a computer or device. A dropper contains an obfuscated malicious binary within itself and "drops" it onto the victim's computer. A worm steals data while propagating across networks. A keylogger tracks the keys typed on the victim's computer. Adware is malicious code that propagates advertisements on a computer device.

In the past five years, the number of malware samples per year registered by AV-TEST Institute has increased over 262%, from 326 million samples detected in 2015 to 856 million samples detected in 2018 (AV-Test, 2018). As the number of devices connected to the Internet continues to increase, there will likely be corresponding increases of the number of malware infections on those devices. Not only has the total amount of malware increased, so has the number of distinct variants (Symantec, 2018).

Due to their simplicity and performance, signatures (distinctive bit patterns of known malware) are the most commonly used way to detect malware. However, signatures can only detect previously identified malware pieces; they have difficulty detecting new malware and variants of old malware with the same functionality (Symantec, 2018). The only way to really be sure about a suspicious binary would be to have it reviewed by an analyst who would likely use a combination of static and dynamic analysis techniques. Static analysis is the technique of analyzing the suspected malware without executing it. The binary can be disassembled to examine the instructions, file headers, program sections, import libraries, and statistical inferences. Authors of malicious binaries often understand static-analysis techniques and try to defeat detection measures using anti-analysis techniques such as obfuscation of the executable code.

Malware analysts can also examine a suspicious binary using dynamic methods within a “sandbox” environment. A sandbox is a separate environment that isolates the executing binary from affecting anything outside the environment. Although very effective, analysis may take days depending on the length and complexity of the binary. Having analysts review every binary that enters the network is not cost-effective. Automating this would require intelligent data analytics and computing ability comparable to the human analyst. Identifying malware is complex; for instance, a malware binary executing on one platform may provide different results than when it executes on a different platform. Therefore, we must construct models for each platform and its intricacies.

This study examined Windows “portable executables” (PE file format). The Windows operating system is widely distributed across the globe and a popular target for attacks. When new files are introduced to a Windows system through downloads or storage transfer, they are usually analyzed by installed security software. This may include anti-virus software residing on the host system or an intrusion-detection system at the point of transfer.

N-grams are a sequences of N successive items of a sample. Statistical analysis on N-grams is often done in static analysis of the byte sequence of a binary. Analysis of sequences is used across a variety of domains including speech recognition, biology, and chemistry; an example in biochemistry is looking for common amino acid strings (Osmanbeyoglu, 2011). For image analysis, the technique has gained much popularity in recent years due to its performance and accuracy; one project used N-grams in gray-scale images to detect objects in images (Bui, Lech, Cheng, Neville, & Burnett, 2016).

Artificial neural networks were first introduced in 1958 by psychologist Frank Rosenblatt (Rosenblatt, 1958). However, they did not garner much attention until their recent success use with image and speech recognition. This has led researchers to experiment with the models and apply them to problems in many fields. In particular, convolutional neural networks have gained much attention for the task of image classification. An example of early convolutional neural networks (Cun, 1994) classified numbers. The network was greatly improved upon during the ImageNet competition in

2012 with the iteration called AlexNet (Krizhevsky, 2017), which added layers of convolution and pooling. It could classify more complex objects and object hierarchies. Convolutional neural networks have been applied mainly to image recognition and natural-language processing. In cybersecurity, neural networks are used to analyze network traffic, system logs, and binaries to detect malicious activity (Kolosnjaji, Zarras, Webster, & Eckert, 2016; Lopez-Martin, 2017).

Static analysis on N-grams of disassembled binaries was performed by Hassen et al. (Hassen, Carvalho, & Chan, 2017). N-grams are a connected sequence of N terms of a sample; in our case, the terms are bytes. The study analyzed the binaries for N-gram features and applied random-forest and logistic-regression machine-learning models. They created frequency arrays for 2-gram, 3-gram, and 4-gram control-statement sequences. In addition, because many 4-gram and some 3-gram sequences were never observed, the study hashed the gram values into a smaller bit space; for example, 2-grams on bytes have 16 bits but were hashed to a 12-bit space. This thesis explored creating a smaller array space by focusing only on N-grams that occur most frequently, outside three standard deviations from the mean, were unique to either malware or benign samples.

Other work (Gong, 2016) that attempted to detect malware in Windows portable executable files took three static approaches to obtaining features. One approach extracted the dynamic-link libraries used by the binary, another extracted particular strings in the binary, and another identified 2-gram sequences from random subsets of the binary. To extract the dynamic link libraries, the researchers use objdump (Stallman, 1984). The terms and their frequency were recorded for all executables. Their analysis resulted in a feature set size of 414 of the most frequent DLL references discovered within their sample of executables. This is done in this thesis except fewer DLL references will be sought. In addition, this thesis also used section names, the total number of sections, and the average entropy across all sections to help distinguish executables. The study's third technique, identifying 2-grams from a random subset of the binary, is also a bag-of-words style feature set. The bag-of-words approach operates by identifying a set of terms and recording the frequency of each term. It is considered a “bag” because the order of the terms does not matter. This thesis also identified 2-grams from the binary in a bag-of-words fashion.

Other work did malware classification by converting a binary into a gray-scale image, and then running it through a K-nearest-neighbors algorithm (Nataraj, 2011). The dataset was comprised of various families of malware without benign samples. The objective was to classify the malware as a backdoor, worm, Trojan, dialer, Trojan downloader, password stealer, or another of the 25 families they studied. The gray-scale image was created by interpreting each byte as a pixel value in the range 0–255; it was sampled, by taking the average of evenly spaced locations, to reach a feature set size of 320, then used as input for the K-nearest neighbor algorithm. The classification reached accuracies above 98%. A similar malware classification on the Microsoft Kaggle classification challenge dataset was done in Gibert (2016) and Microsoft (2015). They modified the gray scale approach by sampling to a feature set size of 1024 to represent a 32 x 32 image where each feature was a gray-scale pixel ranging from 0–255. Then the image was used as input for several convolutional neural network architectures. This work was adapted again for the use of malware detection (Kalash et al., 2018). This thesis sampled to a 1024 feature set with each pixel value ranging from 0–65535 so the gray-scale range was extended to 16 bits instead of the 8 used in the previous studies.

Nvidia, a producer of graphics processing units (GPUs), has published work in malware detection using similar algorithms on GPUs (Raff, 2017). This analyzed the entire binary’s raw bytes and input the information into various neural network architectures. One architecture featured a raw byte embedding strategy and another used a chunking strategy. This thesis took a simpler approach to the architecture and with sampling.

Another study translated the bytes to machine language op codes and library calls as a primitive kind of disassembly before attempting to detect malware (Zak, 2017). They studied three methods: translation to instructions, translation to instruction-parameter types, and translation to function calls (El-Sherei, n.d.). Each N-gram was analyzed, translated, and input to a logistic regression model. 1-gram and 2-gram features were used in all three models; 3-gram and 4-gram features were used in the instruction parameter type and function call translations; and 5-gram and 6-gram features were used for function-call translations.

Attackers often employ obfuscation techniques that only become apparent when the binary is executed. For this reason, researchers have also explored dynamic features such as the sequence of system calls (Kolosnjaji, Zarras, Webster, & Eckert, 2016; Pfoh, Schneider, & Eckert, 2013; Attaluri, Mcghee, & Stamp, 2008). There have also been combinations of static and dynamic analysis using convolutional neural networks and feed-forward neural networks (Kolosnjaji et al., 2017).

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODS

A. DATA PREPARATION

The test set used 5,518 binaries, where 3,251 malware samples were obtained from VirusShare and the other 2,267 were benign samples. All the malware was Windows PE files (VirusShare, 2018). The 2,267 samples labeled as benign were executables that were extracted from classroom computers and the NPS forensic collection DEEP (McCarrin, Gera, Rowe, & Allen, 2017). No benign samples were known malware. The malware obtained from VirusShare were confirmed as malware by several antivirus sources including Symantec, BitDefender, Microsoft, McAfee, and Kaspersky.

The programs that performed the analysis were written in the Python programming language. The development and testing were done on an Ubuntu Linux distribution to avoid accidental contamination of the Windows malware since most malware is unlikely to infect a Linux system. Three separate analysis strategies were used. One converted a binary to a gray-scale image, one used statistical methods, and one identified the dynamic-link libraries and sections of the binaries. The gray-scale method and the statistical method were completed solely within the Linux environment using Python. However, the identification of dynamic-link libraries and sections in the binary used the Cuckoo Sandbox environment (Cuckoo, 2018) set up in a virtual machine using VMware with a Windows 7 Professional operating system.

The binary to gray-scale analysis required preparation of the data. A directory with the test set was the input to the program. Then, each executable in the directory was read byte-by-byte as a binary number. A “summary” image of 32 x32 pixels, or 1024 bytes, was calculated by averaging the bytes within the sub-windows, where the window size was proportional to the size of the file. Each 32 x 32 array was attached to a larger array containing all the file summaries in a directory. The processing flow is displayed in Figure 1.

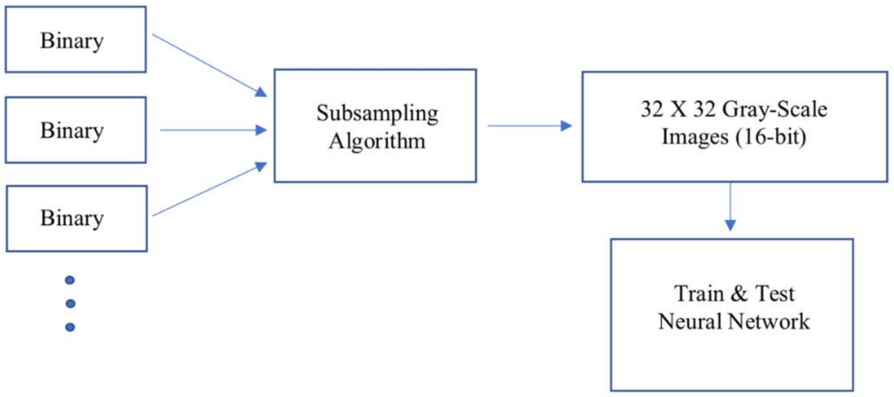
Binary
Binary
Binary
Subsampling Algorithm
32 X 32 Gray-Scale Images (16-bit)
Train & Test Neural Network

Figure 1. The sequence of steps from the binary samples to training the neural network using subsampled gray-scale images.

When a binary file was opened, if its size was less than 2048, no subsampling occurred. Instead, every two bytes were treated as a 16-bit integer value, and the remaining spaces were assigned zeros. If the initial size is greater than 2048, the size was divided by 2048 and the result ignoring any fractional remainder was the number of 16-bit values that were averaged together. For example, a modulo result from a file size of 5,000 and the modulo parameter of 2,048 is 2.44. Truncating the decimal results in the integer 2 as the subsampling variable, so every two 16-bit values were averaged together and the remaining 904 bytes were dropped. The result is an array of 1024 16-bit values that is used as input for the neural network.

The statistical method analyzed the binaries to extract N-grams that were input for machine learning. N-grams are N consecutive bytes in a sample. A feature-set array was created containing particular N-grams that were statistically significant for malware in the data. The data samples were split into two subsets, malware and benign. Four dictionaries for each subset were used to record the frequency of 1-grams, 2-grams, 3-grams, and 4-grams. A scoring system was used to choose the N-grams added to the feature set using the distance of the malware fraction from the expected value in units of standard deviation calculated within each dictionary. N-grams with scores greater than three standard deviations from the mean were stored in an array we will call the feature set. Originally, the feature set had 32,768 N-grams which was far too large for processing. To reduce it, 3-grams were limited to the top 50 scores and 4-grams were limited to the top 100 scores. The feature set was reduced to a total of 185 features to avoid overfitting. Of the 185 chosen, thirty-five were 2-grams, fifty were 3-grams, one-hundred were 4-grams, and no 1-grams.

To analyze a binary, the program extracted each N-gram in the binary and recorded its existence with the integer 1 in a separate array of length equal to the feature set array. Each binary was thus represented by a feature array of 1's and 0's. The processing sequence can be seen in Figure 2.

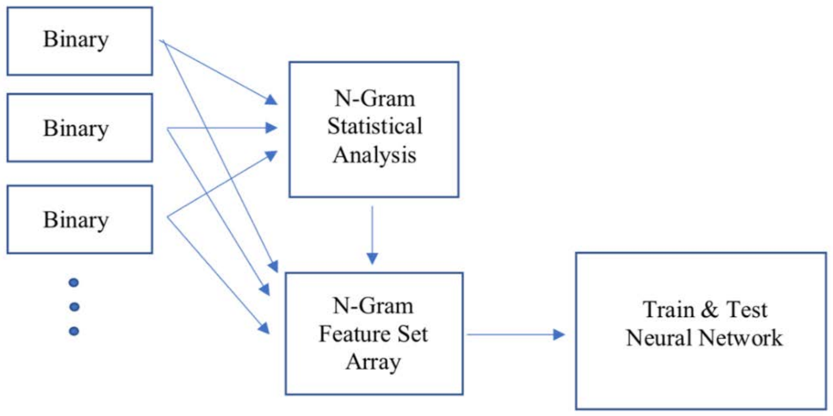

Figure 2. The sequence of steps from binary samples to training the neural network using n-gram features.

Identification of dynamic-link libraries and sections within the binary was done with the Cuckoo sandbox environment (Cuckoo, 2018). The Cuckoo agent was installed on the Windows 7 system on a VMware virtual machine (VMware, 2018); Ubuntu was the base operating system used. The Cuckoo package "exe" was used to analyze each sample binary (Automated Process, 2015). Its output was a JSON-formatted file reporting the dynamic-link libraries identified, the sections, the entropy for each section, and other details. The names in JSON reports were counted and stored in a Python dictionary. The ten most frequently reported names were included in the feature set for each binary. In addition, the average entropy across all sections and the number of sections in the binary were used as features. The process flow is given in Figure 3.

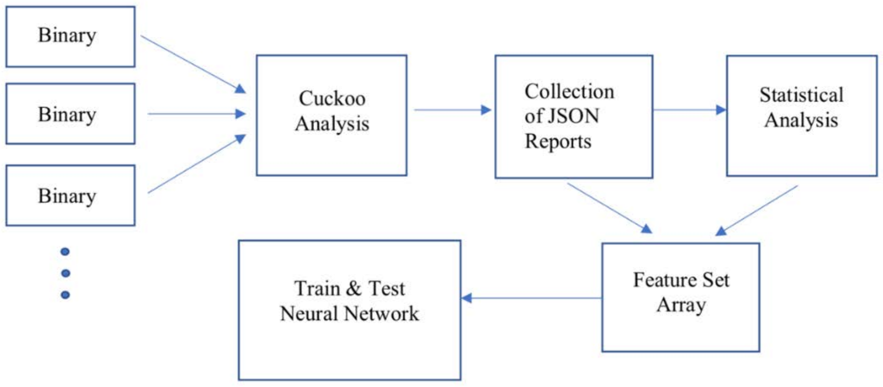

Figure 3. The sequence of steps from binary samples to training the neural network using dynamic link library and section data.

B. NEURAL-NETWORK APPLICATION

At the lowest level, an artificial neuron simulates a biological neuron where the dendrites accept an input and the axon outputs a signal (Rosenblatt, 1958). The neurons are structured into a network of layers. Typically, a neural network consists of an input layer, hidden layer(s), and the output layer. Each layer is fully-connected which means nodes (neurons) connect with every node from the previous and subsequent layer. The strength of the connection between the nodes is a weight that is determined through the training process. Usually, the training process uses backpropagation to adjust the weighted values to improve closeness to correct output. Figure 4 shows a typical plan.

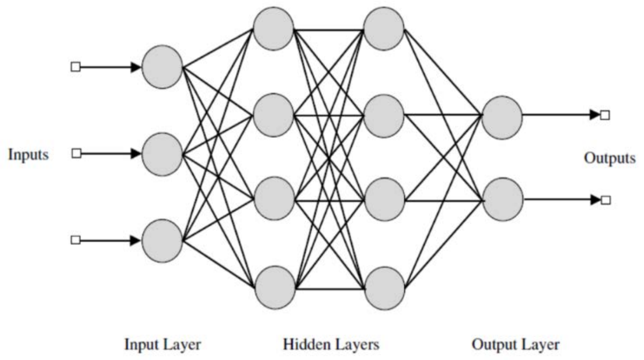

Figure 4. Feed-forward neural network with fully-connected layers. Source: IIIT-H Virtual Labs (2018).

An enhanced neural-network structure is the convolutional neural network (Cun, 1994). This structure is common for image classification and was used in our strategy that uses gray-scale images converted from binaries. When used for image classification, its hidden layers are a combination of convolutional and "pooling" layers followed by a "dense" layer. The convolution layer is paired with an activation function and the output layer is paired with a logistic function to provide a result between 0–1 for each node in the output layer. A convolution is an important technique used in signal processing and is defined as an inner product of two signals that form an output signal (Smith, 2012). The two incoming signals are the input signal and the impulse signal. For image classification, a convolution is where a digital filter (a kernel) is mapped to data and corresponding items are multiplied. There are three components in the process: the input data, kernel filter, and the output data. For images, the kernel filter is the impulse signal in the form of a square matrix. The kernel filter is laid "on top" of the input with the center of the filter corresponding to the location of the result in the output image, as shown in Figure 5.

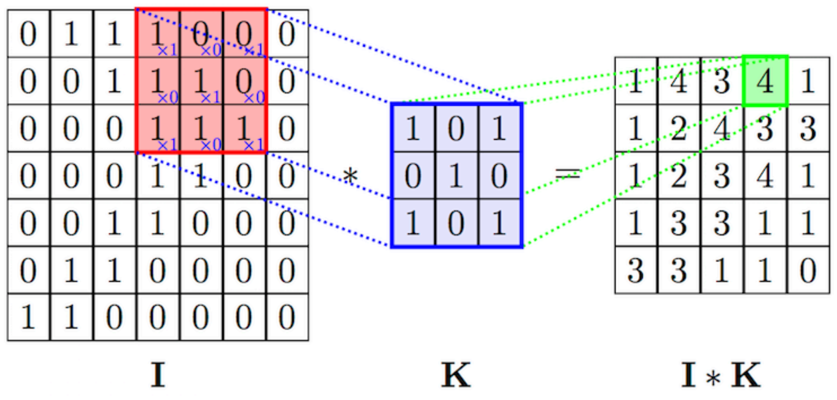

Figure 5. Example convolution where the kernel filter, *K*, is laid on top of the image, *I*, resulting in image, *I*K*. Source: Spark (2017).

To ensure the result size has the same size as the input, padding is usually added to the input prior to convolution in the form of a border of zeroes around the entire image. The convolution operation (layer) is usually followed by an activation function that normalizes it. This thesis used the rectified linear unit activation function "ReLU," which replaces negative numbers with a 0 and retains values greater than or equal to 0. The pooling layer is designed to reduce the data size in the number of nodes and parameters. The size of reduction is dependent on the size of the pooling filter and the “stride.” The pooling filter is like a border that surrounds a portion of the image, and some function is applied to the values within the designated border. In this study, the max pooling method was used which results in the maximum value within the border of a set of values. In the case of RGB images, each pixel contains three values, one for each color. Therefore, we can visualize the images as three layers of pixel values. Max pooling would output the maximum value within the set values contained in the specified border for each color layer. For our gray-scale images, a single value represents a pixel and therefore max pooling operates on the single layer. Thus, with a pooling filter size of 2 × 2, a set of four values are compared to determine the maximum value. The maximum value from each set is carried to the next layer. The stride determines how many pixels the pooling filter shifts between operations. For example, a 2 × 2 max pooling filter applied to a 4 × 4 image with a stride of 2 would produce the output image shown in Figure 6.

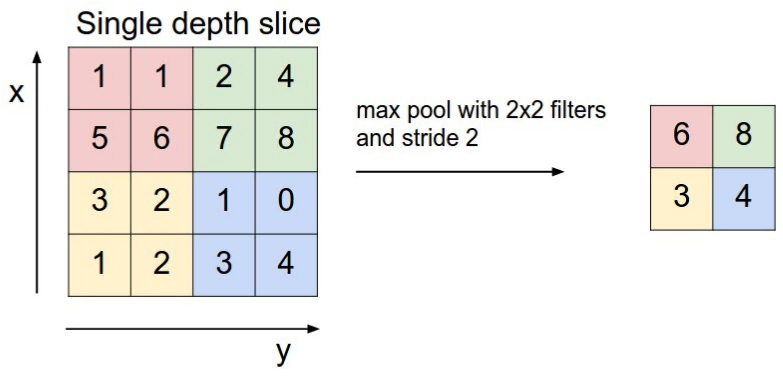

Figure 6. Max pooling with filter size of 2 × 2, and stride of 2, applied to a 4 × 4 image. Li (2018).

The dense or “fully-connected” layer is where each node from the previous layer connects to every node in the next layer (Figure 7). This layer is used in all three architectures.

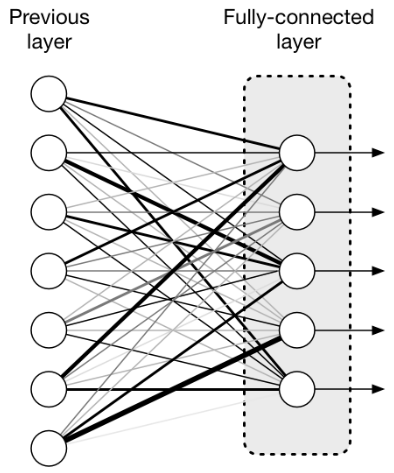

Figure 7. Example of a fully-connected layer. Source: Hollemans (2017).

The output layer makes the classification. This thesis classifies binaries as malicious or benign, so there are two output neurons. The likelihood of each node is normalized using the logistic function softmax shown in Figure 8.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Figure 8. Softmax equation where K is the number of nodes in the output layer, and z is the value at that particular node.

The result is a categorical probability distribution represented by values between 0–1, at each output node. The sum of all output nodes equals 1. The output layer node with the largest value is the class that the neural network predicts. Training does backpropagation adjusting the weights to make a better prediction the next time.

C. ARCHITECTURES

A separate neural network architecture was used for each of the three analysis strategies. The binary-to-gray image method used a convolutional neural network and the other two strategies used a feed-forward network. The samples were split into 80% training and 20% test sets were used with 10-fold validation methods, meaning the samples were randomly selected ten ways to avoid overfitting. The binary to gray-scale image architecture used a 2-dimensional convolutional neural network similar to one used in an example on the TensorFlow website (TensorFlow, 2018). This network design was chosen due to its simplicity and previous success with gray-scale images. The neural network had two convolutional layers, each followed by a pooling layer, and the final two layers were the dense layer and the output layer. The first layer used a kernel size of 5×5, 2 border units of padding, 32 filters, and a ReLU activation layer. There were 32,768 neurons in this layer and each one contained 25 inputs. Next, there was the pooling layer with pool size of 2×2 and stride of 2. This pooling layer reduced the image size to 16 x16 and the size of the layer was 8,192 neurons, each with four inputs. The third layer was the second convolutional layer with the same activation and padding but with a kernel size of 3×3 and 64 filters. There were 16,384 neurons in this layer and each contained nine inputs. This was followed by a pooling layer identical to the previous one, reducing each image to 8×8, and the layer size was 4,096 neurons, each with four inputs. A dense layer followed with 1,024 neurons with 4,096 inputs each. The final layer was two output neurons with 1,024 inputs each. The model was trained using a gradient-descent optimizer at a learning rate of 0.01. The neural-network architecture for the N-gram strategy was a feed-forward model with an input layer taking input of the feature set of 185. It contained two hidden layers with sizes 16 and 4, respectively. In a feed-forward model, each neuron was connected to each neuron in the previous layer. The output layer was two nodes that predicted whether the input sample was malware or not. The model was trained using the Adam stochastic optimization algorithm (Kolkiewicz, 2010).

The neural-network architecture for the dynamic-link library strategy was a feed-forward model with an input layer having inputs of the feature set of twelve items. It had two hidden layers of sizes 6 and 3, respectively. Each neuron in the first layer had twelve inputs and second-layer neurons had six inputs. Like the other models, the output layer had two neurons that predicted whether the input sample was malware or not. The model was trained using the same Adam stochastic optimization algorithm used in the N-gram strategy (Kolkiewicz, 2010). The feed-forward architectures were implemented using the Python Sci-kit Learn library (Sci-kit Learn, 2018).

IV. RESULTS

The binary-to-grayscale, statistical-N-gram, and dynamic-link-library strategies were tested with the previously described neural-network architectures. The binary-to-grayscale strategy used the convolutional neural network architecture, while both the statistical-N-gram and dynamic-link-library strategies used a feed-forward neural network with different numbers of nodes per layer. All three architectures had two output neurons, one for each designated class, that output a probability value. The larger probability of the two output neurons identifies the prediction. The preprocessed inputs were split 80% for testing and 20% for training the models and used with the 10-fold validation method (that is, with ten random selections of the 80%–20% split).

The models were evaluated using error-matrix metrics, overall accuracy, and average probabilities of the output neurons during testing. The error-matrix metrics included true positives, true negatives, false positives, false negatives, precision, recall, and f-score. The neural networks were designed such that malware was designated as a positive prediction (1), and the benign binary was designated as a negative prediction (0). If the neural-network model predicts the binary to be malware (1) and the actual classification is malware, it is counted as a true positive. If the prediction is malware and the true classification is a benign binary, it is a false positive. If the prediction was a benign binary (0) and the true classification is malware, it is a false negative. These three metrics provide the parameters for calculating precision, recall and f-score. Precision is the fraction of correct predictions of all predictions. Recall is the fraction of correct predictions of all positive samples identified. The f-score is a measure of accuracy that considers both the precision and recall. The equation is displayed in Figure 9. The generic accuracy, total number of samples correctly predicted in either direction divided by the total number of tested samples is also used to evaluate the data.

$$F = 2 \times \frac{Precision \times Recall}{Precision \ + Recall}$$

Figure 9. F-score accuracy equation.

Overall, the dynamic-link-library strategy and the N-gram analysis performed similarly by correctly predicting 90% of the samples. The gray-scale image strategy performed poorly in recognizing only 56% of the samples.

A. STRATEGY I: BINARY TO 16-BIT GRAY-SCALE IMAGE

The strategy of converting a binary to a 16-bit gray-scale image of size 32 × 32, from Figure 1, did not perform well. As shown in Table 1, it produced an accuracy of 56%. Its recall is 100% but that is because no samples were predicted as benign: Every input was predicted as malware. In previous work, the convolutional neural networks used for malware classification used 8-bit pixel sizes (Kalash et al., 2018; Karaj, 2011) rather than 16 bits in our work. Our bit space per pixel was 256 times greater than in the previous work. The increased possibilities and the averaging caused by subsampling may have caused insufficient information for the neural network to learn.

Table 1. Accuracy results for the gray-scale-image strategy.

Precision	Recall	F-score	Accuracy
55.91%	100%	71.72%	55.89%

B. STRATEGY II: STATISTICAL N-GRAMS

The strategy of identifying particular N-grams, from Figure 2, predicted the correct classification for the binary sample an average of 89% of the time with just over 1% standard deviation between the test runs as shown in Table 2. The precision is recognizably lower than the recall of the model. This means the model identified most malware samples but did so while incorrectly predicting a considerable portion of non-malware as malware, also known as false positives. Table 4 shows the average probabilities of each output neuron. The rows indicate the status of the predictions. When the neural network predicted correctly, the probability of the correct neuron is high, relative to incorrect predictions, and has a smaller percentage of deviation than when the neural network predicted incorrectly. Also, when predicting correctly, the average output probability of the malware neuron is higher than the benign neuron.

Table 2. Accuracy results of the statistical-N-gram strategy

Precision	Recall	F-score	Accuracy
88.01%± 1.32	93.55%± 3.13	90.06%± 1.22	89.24%± 1.15

Table 3. Confusion matrix of one of the ten test runs for the statistical-N-gram strategy.

	Malware (Actual)	Not Malware (Actual)
Malware (Predicted)	607	88
Not Malware (Predicted)	33	434

Table 4. Average probabilities at each output neuron in the N-gram strategy

	Benign Neuron (0)	Malware Neuron (1)
True Positive (TP)	14.4% ± 11%	85.5% ± 11%
True Negative (TN)	94.5% ± 10%	5.4% ± 10%
False Positive (FP)	26.0% ± 13%	73.9% ± 13%
False Negative (FN)	73.6% ± 17%	26.3% ± 17%

C. STRATEGY III: DYNAMIC LINK LIBRARIES AND SECTIONS

The accuracy of the dynamic-link-library strategy, from Figure 3, performed similarly to the previous strategy in correctly classifying an average of 90% of the binary samples as shown in Table 5. The precision is noticeably higher than the recall. This means that when the model predicts the binary as malware, it is often correct. However, it does not properly identify more than 13% of the malware samples. The output neuron probabilities in Table 7 are similar to those in Table 4 such that the probabilities are relatively high, with little deviation, when correctly predicting the classification. The difference is that, when correctly predicting the class, the malware neuron averages a higher probability.

Table 5. Accuracy results of the dynamic-link-library strategy

Precision	Recall	F-score	Accuracy
93.89%± 1.16	86.75%± 1.01	90.17%± 0.61	89.66% ± 0.65

Table 6. Confusion matrix

	Malware (Actual)	Not Malware (Actual)
Malware (Predicted)	532	45
Not Malware (Predicted)	73	467

Table 7. Average probabilities of each output neuron in the dynamic-link-library strategy

	Benign Neuron (0)	Malware Neuron (1)
True Positive (TP)	5.8% ± 9%	94.1% ± 9%
True Negative (TN)	87.1% ± 11%	12.8% ± 11%
False Positive (FP)	29.8% ± 16%	70.1% ± 16%
False Negative (FN)	75.8% ± 11%	24.1% ± 11%

V. CONCLUSION

A. FINDINGS

Two strategies performed at an accuracy of around 90% while the third strategy performed poorly. The binary-to-grayscale image-conversion strategy lost information during the conversion process and extracted too-sparse data, contributing to its poor performance. Large binaries are very susceptible to information loss; for example, a binary of size 8 KB would require averaging four 16-bit values, and the average is not often unique.

The statistical-N-gram strategy and the dynamic-link-library strategy had overall similar performance but had different performance in precision and recall. The N-gram strategy resulted in a better recall than precision whereas the dynamic-link-library strategy had better recall. The probabilities of the output neurons in the two strategies were higher, with less deviation, when the model correctly predicted the class. Although the strategies performed at around 90% in accuracy, previous work found better-performing strategies (Kolosnjaji et al., 2017; Raff, 2017). Raff et al. were able to achieve a 94% accuracy using their set of 2 million binary samples. Using a hybrid neural network consisting of feed-forward and convolutional architectures, Kolosnjaji et al. achieved a classification accuracy of 92%.

B. FUTURE WORK

Our results were encouraging, but further improvements are possible. The number of features and the hidden-layer sizes can be adjusted for the two feed-forward neural-network strategies. However, the possibility of overfitting must be considered. In particular, the number of weights between the inputs and the first layer must never exceed the number of data samples, so increasing the number of data samples would be beneficial.

Because the precision and recall for the two feed-forward neural-network strategies are opposite of one another, it could be useful to combine the two strategies in some way such as adding their outputs or combining them with additional neurons.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX. NEURAL NETWORK ARCHITECTURES

```
################# Start Binary-to-grayscale 16-bit PART 1  ####################

#!/usr/bin/python

import numpy as np
import matplotlib.pyplot as plt
import os
import struct
import pickle

BENIGN=True
MALWARE=False

if BENIGN:
    benign_dir ='/home/nps/Desktop/CNN_Malware_Analysis/Binaries/'
    directory=benign_dir
    output_value=0
    pickle_in="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bin2img_B_input"
    pickle_out="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bin2img_B_output"
    print('BENIGN START')

if MALWARE:
    malware_dir = '/home/nps/Documents/Malware/Virus.Win/'
    directory=malware_dir
    output_value=1
    pickle_in="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bin2img_M_input"
    pickle_out="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bin2img_M_output"
    print('MALWARE START')


vectors16=[]
v16=np.empty([32,32], dtype=int)
v16x=[]
output_arr=[]


for filename in os.listdir(directory):
    full_path = directory + filename
    if os.path.isfile(full_path):
        with open(full_path,"rb") as binary_file:
            del vectors16[:]
            data=binary_file.read()
            file_size=os.stat(full_path).st_size
            modulo_result=file_size%2048
            block_size=file_size//2048
            binary_file.seek(0)
            if block_size==0:
                block_size=1

            for v in range(1024):
                avg=0
                for x in range(block_size):
                    bytes16=binary_file.read(1)
                    if len(bytes16) < 1:
                        break
                    g1=ord(struct.unpack('c',bytes16)[0])*256
                    bytes16=binary_file.read(1)
                    if len(bytes16) < 1:
                        break
                    avg=ord(struct.unpack('c',bytes16)[0]) + g1

                col=v%32
                row=v//32
                vectors16.append(avg/block_size)
                v16[row,col]=avg/block_size


            nvectors16=np.array(vectors16)
            v16x.append(np.copy(nvectors16))
            output_arr.append(output_value)

npVec16=np.array(v16x)
out_arr=np.array(output_arr)

pickle.dump(npVec16, open(pickle_in, "wb"))
pickle.dump(out_arr, open(pickle_out, "wb"))


################ Binary-to-grayscale 16-bit PART 2  ########################


#!/usr/bin/python

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from sklearn.model_selection import train_test_split
```

```
import numpy as np
import tensorflow as tf
import pickle

tf.logging.set_verbosity(tf.logging.INFO)

def cnn_model_fn(features, labels, mode):

  input_layer = tf.reshape(features["x"], [-1, 32, 32, 1])

  # Convolutional Layer #1
  conv1 = tf.layers.conv2d(
    inputs=input_layer,
    filters=32,
    kernel_size=[5, 5],
    padding="same",
    activation=tf.nn.relu)

  # Pooling Layer #1
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)

  # Convolutional Layer #2 and Pooling Layer #2
  conv2 = tf.layers.conv2d(
    inputs=pool1,
    filters=64,
    kernel_size=[3, 3],
    padding="same",
    activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)

  # Dense Layer
  pool2_flat = tf.reshape(pool2, [-1, 8 * 8 * 64])
  dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
  dropout = tf.layers.dropout(
    inputs=dense, rate=0.4, training=mode == tf.estimator.ModeKeys.TRAIN)

  # Logits Layer
  logits = tf.layers.dense(inputs=dropout, units=2)

  predictions = {
    "classes": tf.argmax(input=logits, axis=1),
    "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
  }
```

```
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

    loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

    if mode == tf.estimator.ModeKeys.TRAIN:
        optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
        train_op = optimizer.minimize(
            loss=loss,
            global_step=tf.train.get_global_step())
        return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

    eval_metric_ops = {
      "accuracy": tf.metrics.accuracy(
          labels=labels, predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(
        mode=mode, loss=loss, eval_metric_ops=eval_metric_ops)


def main():

    benign_data =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_image/bi
n2img_B_input", "rb"))
    benign_labels =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_image/bi
n2img_B_output", "rb"))
    malware_data =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_image/bi
n2img_M_input", "rb"))
    malware_labels =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_image/bi
n2img_M_output", "rb"))

    input_array = np.concatenate((benign_data, malware_data), axis=0)
    output_array = np.concatenate((benign_labels, malware_labels), axis=0)

    input_df=pd.DataFrame(input_array)
    output_df=pd.DataFrame(output_array)

    cross_val_num=10

    for i in range(cross_val_num):
        X_train, x_test, Y_train, y_test = train_test_split(input_df, output_df, test_size =
0.2, shuffle=True)

        train_data=np.asarray(X_train)
        train_labels=np.asarray(x_test)
        eval_data=np.asarray(Y_train)
        eval_labels=np.asarray(y_test)

        bin2img_classifier = tf.estimator.Estimator(
        model_fn=cnn_model_fn,
model_dir="/home/nps/Desktop/CNN_Malware_Analysis/models/bin2img_convnet_mo
del")

        tensors_to_log = {"probabilities": "softmax_tensor"}
        logging_hook = tf.train.LoggingTensorHook(
            tensors=tensors_to_log, every_n_iter=50)

    # Training
        train_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": train_data},
            y=train_labels,
            batch_size=100,
            num_epochs=None,
            shuffle=True)
        bin2img_classifier.train(
            input_fn=train_input_fn,
            steps=20000,
            hooks=[logging_hook])

    # Testing
        eval_input_fn = tf.estimator.inputs.numpy_input_fn(
            x={"x": eval_data},
            y=eval_labels,
            num_epochs=1,
            shuffle=False)
        eval_results = bin2img_classifier.evaluate(input_fn=eval_input_fn)
        print(eval_results)

if __name__ == "__main__":
    main()


############  START STATISTICAL N_GRAM ANALYSIS PART 1 ############
```

```
#!/usr/bin/python

import numpy as np
import os
import struct
import pickle
from collections import Counter

benign_dir ='/home/nps/Desktop/CNN_Malware_Analysis/Binaries/'
malware_dir = '/home/nps/Documents/Malware/Virus.Win/'

gram2_run=True
gram3_run=True
gram4_run=True

byte_hist1_S3={}
byte_hist2_S3={}
byte_hist3_S3={}
byte_hist4_S3={}

byte_hist1_B={}
byte_hist2_B={}
byte_hist3_B={}
byte_hist4_B={}

for b in range(2^8):
    byte_hist1_B[int(b)]=0

if gram2_run:
    for b in range(2^16):
        byte_hist2_B[int(b)]=0

if gram3_run:
    for b in range(2^24):
        byte_hist3_B[int(b)]=0

if gram4_run:
    for b in range(2^32):
        byte_hist4_B[int(b)]=0

print('BENIGN START')
for filename in os.listdir(benign_dir):
    full_path = benign_dir + filename
    if os.path.isfile(full_path):
        with open(full_path,"rb") as binary_file:
```

```
        data=binary_file.read()

        gram1=0
        gram2=0
        gram3=0
        gram4=0

        for byte in data:
           gram1=ord(struct.unpack('c',byte)[0])

           if gram1 in byte_hist1_B:
                   byte_hist1_B[gram1]+=1
           else:
                   byte_hist1_B[gram1]=1
           if gram2_run:
                   g2 = gram1 + gram2*256
                   if g2 in byte_hist2_B:
                           byte_hist2_B[g2]+=1
                   else:
                           byte_hist2_B[g2]=1
           if gram3_run:
                   g3 = gram1 + gram2*256 + gram3*256*256
                   if g3 in byte_hist3_B:
                           byte_hist3_B[g3]+=1
                   else:
                           byte_hist3_B[g3]=1
           if gram4_run:
                   g4 = gram1 + gram2*256 + gram3*256*256 + gram4*256*256*256
                   if g4 in byte_hist4_B:
                           byte_hist4_B[g4]+=1
                   else:
                           byte_hist4_B[g4]=1

           gram4 = gram3
           gram3 = gram2
           gram2 = gram1

        binary_file.close()


print('MEANS/STD BENIGN')
gram1_meanB = np.mean(byte_hist1_B.values())
gram1_stdB = np.std(byte_hist1_B.values())
print('GRAM 1 Mean/std')
print(gram1_meanB)
print(gram1_stdB)

if gram2_run:
    gram2_meanB = np.mean(byte_hist2_B.values())
    gram2_stdB = np.std(byte_hist2_B.values())
    print('GRAM 2 Mean/std')
    print(gram2_meanB)
    print(gram2_stdB)

    for key, value in byte_hist2_B.items():
        byte_hist2_S3[key]=((value-gram2_meanB)/gram2_stdB)*10

if gram3_run:
    gram3_meanB = np.mean(byte_hist3_B.values())
    gram3_stdB = np.std(byte_hist3_B.values())
    print('GRAM 3 Mean/std')
    print(gram3_meanB)
    print(gram3_stdB)

    for key, value in byte_hist3_B.items():
        byte_hist3_S3[key]=((value-gram3_meanB)/gram3_stdB)*10

if gram4_run:
    gram4_meanB = np.mean(byte_hist4_B.values())
    gram4_stdB = np.std(byte_hist4_B.values())
    print('GRAM 4 Mean/std')
    print(gram4_meanB)
    print(gram4_stdB)

    for key, value in byte_hist4_B.items():
        byte_hist4_S3[key]=((value-gram4_meanB)/gram4_stdB)*10

byte_hist1_B.clear()
byte_hist2_B.clear()
byte_hist3_B.clear()
byte_hist4_B.clear()

###   END BENIGN ANALYSIS   ###
### START MALWARE ANALYSIS ###


byte_hist1_M={}
byte_hist2_M={}
byte_hist3_M={}
byte_hist4_M={}
```

```
print('MALWARE START')
for filename in os.listdir(malware_dir):
    full_path = malware_dir + filename
    if os.path.isfile(full_path):
        with open(full_path,"rb") as binary_file:
            data=binary_file.read()

            gram1=0
            gram2=0
            gram3=0
            gram4=0

            for byte in data:
                gram1=ord(struct.unpack('c',byte)[0])

                if gram1 in byte_hist1_M:
                        byte_hist1_M[gram1]+=1
                else:
                        byte_hist1_M[gram1]=1
                if gram2_run:
                        g2 = gram1 + gram2*256
                        if g2 in byte_hist2_M:
                                byte_hist2_M[g2]+=1
                        else:
                                byte_hist2_M[g2]=1
                if gram3_run:
                        g3 = gram1 + gram2*256 + gram3*256*256
                        if g3 in byte_hist3_M:
                                byte_hist3_M[g3]+=1
                        else:
                                byte_hist3_M[g3]=1
                if gram4_run:
                        g4 = gram1 + gram2*256 + gram3*256*256 + gram4*256*256*256
                        if g4 in byte_hist4_M:
                                byte_hist4_M[g4]+=1
                        else:
                                byte_hist4_M[g4]=1

                gram4 = gram3
                gram3 = gram2
                gram2 = gram1

    binary_file.close()
```

```
print('MEANS/STD MALWARE')
print('GRAM 1 Mean/std')
gram1_meanM = np.mean(byte_hist1_M.values())
gram1_stdM = np.std(byte_hist1_M.values())
print(gram1_meanM)
print(gram1_stdM)

if gram2_run:
    gram2_meanM = np.mean(byte_hist2_M.values())
    gram2_stdM = np.std(byte_hist2_M.values())
    print('GRAM 2 Mean/std')
    print(gram2_meanM)
    print(gram2_stdM)

    for key, value in byte_hist2_M.items():
        if key in byte_hist2_S3:
            if (byte_hist2_S3[key]/10)/((value-gram2_meanM)/gram2_stdM)>1:
                byte_hist2_S3[key]=(byte_hist2_S3[key]/10)/((value-
gram2_meanM)/gram2_stdM)
            else:
                byte_hist2_S3[key]=((value-
gram2_meanM)/gram2_stdM)/(byte_hist2_S3[key]/10)
        else:
            byte_hist2_S3[key]=10

if gram3_run:
    gram3_meanM = np.mean(byte_hist3_M.values())
    gram3_stdM = np.std(byte_hist3_M.values())
    print('GRAM 3 Mean/std')
    print(gram3_meanM)
    print(gram3_stdM)

    for key, value in byte_hist3_M.items():
        if key in byte_hist3_S3:

            if (byte_hist3_S3[key]/10)/((value-gram3_meanM)/gram3_stdM)>1:
                byte_hist3_S3[key]=(byte_hist3_S3[key]/10)/((value-
gram3_meanM)/gram3_stdM)
            else:
                byte_hist3_S3[key]=((value-
gram3_meanM)/gram3_stdM)/(byte_hist3_S3[key]/10)
        else:
            byte_hist3_S3[key]=10

if gram4_run:
    gram4_meanM = np.mean(byte_hist4_M.values())
    gram4_stdM = np.std(byte_hist4_M.values())
    print('GRAM 4 Mean/std')
    print(gram4_meanM)
    print(gram4_stdM)

    for key, value in byte_hist4_M.items():

        if key in byte_hist4_S3:
            if (byte_hist4_S3[key]/10)/((value-gram4_meanM)/gram4_stdM)>1:
                byte_hist4_S3[key]=(byte_hist4_S3[key]/10)/((value-
gram4_meanM)/gram4_stdM)
            else:
                byte_hist4_S3[key]=((value-
gram4_meanM)/gram4_stdM)/(byte_hist4_S3[key]/10)
        else:
            byte_hist4_S3[key]=10

byte_hist1_M.clear()
byte_hist2_M.clear()
byte_hist3_M.clear()
byte_hist4_M.clear()

###    END MALWARE ANALYSIS    ###
### START STATISTICAL ANALYSIS ###

pickle_hist2_3={}
pickle_hist3_3={}
pickle_hist4_3={}

if gram2_run:
    g2_mean = np.mean(byte_hist2_S3.values())
    g2_std = np.std(byte_hist2_S3.values())

    for key, value in byte_hist2_S3.items():
        if value>(g2_mean+3*g2_std) or value<(g2_mean-3*g2_std):
            pickle_hist2_3[key]=1

if gram3_run:
    g3_mean = np.mean(byte_hist3_S3.values())
    g3_std = np.std(byte_hist3_S3.values())

    vc = Counter(byte_hist3_S3.itervalues())
    for key, value in vc.most_common(50):
            pickle_hist3_3[key]=1
```

```
if gram4_run:
    g4_mean = np.mean(byte_hist4_S3.values())
    g4_std = np.std(byte_hist4_S3.values())

    vc = Counter(byte_hist4_S3.itervalues())
    for key, value in vc.most_common(100):
            pickle_hist4_3[key]=1

pickle_hist2_3std="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bb_v2_hist2_3s
td"
pickle_hist3_3std="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bb_v2_hist3_5
0"
pickle_hist4_3std="/home/nps/Desktop/CNN_Malware_Analysis/pickles/bb_v2_hist4_1
00"

if gram2_run:
    pickle.dump(pickle_hist2_3, open(pickle_hist2_3std, "wb"))
if gram3_run:
    pickle.dump(pickle_hist3_3, open(pickle_hist3_3std, "wb"))
if gram4_run:
    pickle.dump(pickle_hist4_3, open(pickle_hist4_3std, "wb"))

################ STATISTICAL N_GRAM ANALYSIS PART 2 ###############

#!/usr/bin/python

import numpy as np
import os
import struct
import pickle

gram2_run=True
gram3_run=True
gram4_run=True

benign_dir ='/home/nps/Desktop/CNN_Malware_Analysis/Binaries/'
malware_dir = '/home/nps/Documents/Malware/Virus.Win/'

hist2_3std =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bb_
v2_hist2_3std", "rb"))
hist3_3std =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bb_
v2_hist3_50", "rb"))
```

```
hist4_3std =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bb_
v2_hist4_100", "rb"))

sample=[]
inputs=[]
output_arr=[]

byte_hist1_B={}
byte_hist2_B={}
byte_hist3_B={}
byte_hist4_B={}

BENIGN=True
MALWARE=False
output_value=-1

if BENIGN:
   benign_dir ='/home/james/Desktop/CNN_Malware_Analysis/Binaries/'
   directory=benign_dir
   output_value=0

pickle_in="/home/james/Desktop/CNN_Malware_Analysis/pickles/binarybytesCount_v2
2_B_input"

pickle_out="/home/james/Desktop/CNN_Malware_Analysis/pickles/binarybytesCount_v
22_B_output"
   print('BENIGN START')

if MALWARE:
   malware_dir = '/home/james/Documents/Malware/Virus.Win/'
   directory=malware_dir
   output_value=1

pickle_in="/home/james/Desktop/CNN_Malware_Analysis/pickles/binarybytesCount_v2
2_M_input"

pickle_out="/home/james/Desktop/CNN_Malware_Analysis/pickles/binarybytesCount_v
22_M_output"
   print('MALWARE START')

for filename in os.listdir(directory):
   full_path = directory + filename
   if os.path.isfile(full_path):
      with open(full_path,"rb") as binary_file:
```

```
del sample[:]
byte_hist1_B.clear()
byte_hist2_B.clear()
byte_hist3_B.clear()
byte_hist4_B.clear()
data=binary_file.read()
gram1=0
gram2=0
gram3=0
gram4=0
count = 0

for byte in data:
    gram1=ord(struct.unpack('c',byte)[0])
    count += 1
    if gram1 in byte_hist1_B:
                byte_hist1_B[gram1]+=1
    else:
                byte_hist1_B[gram1]=1
    if count>2:
        if gram2_run:
                    g2 = gram1 + gram2*256
        if g2 in byte_hist2_B:
                        byte_hist2_B[g2]+=1
        else:
                        byte_hist2_B[g2]=1
    if count>3:
        if gram3_run:
                    g3 = gram1 + gram2*256 + gram3*256*256
        if g3 in byte_hist3_B:
            byte_hist3_B[g3]+=1
        else:
            byte_hist3_B[g3]=1
    if count>4:
        if gram4_run:
                    g4 = gram1 + gram2*256 + gram3*256*256 + gram4*256*256*256
        if g4 in byte_hist4_B:
            byte_hist4_B[g4]+=1
        else:
            byte_hist4_B[g4]=1

    gram4 = gram3
    gram3 = gram2
    gram2 = gram1
```

```
        binary_file.close()

        for key,value in sorted(hist2_3std.items()):
          if key in byte_hist2_B:
            sample.append(byte_hist2_B[key])
          else:
            sample.append(0)
        for key,value in sorted(hist3_3std.items()):
          if key in byte_hist3_B:
            sample.append(byte_hist3_B[key])
          else:
            sample.append(0)

        for key,value in sorted(hist4_3std.items()):
          if key in byte_hist4_B:
            sample.append(byte_hist4_B[key])
          else:
            sample.append(0)

        nsample=np.array(sample)
        inputs.append(np.copy(nsample))
        output_arr.append(output_value)

np_inputs=np.array(inputs)
np_output=np.array(output_arr)
pickle.dump(np_inputs, open(pickle_in, "wb"))
pickle.dump(np_output, open(pickle_out, "wb"))

################ STATISTICAL N_GRAM ANALYSIS PART 3 ###############

#!/usr/bin/python

import numpy as np
import pandas as pd
import seaborn as sns
import pickle
import matplotlib.pyplot as plt

from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn import cross_validation
```

```
benign_data =
pickle.load(open("/home/james/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bi
narybytesCount_v22_B_input", "rb"))
benign_labels =
pickle.load(open("/home/james/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bi
narybytesCount_v22_B_output", "rb"))
malware_data =
pickle.load(open("/home/james/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bi
narybytesCount_v22_M_input", "rb"))
malware_labels =
pickle.load(open("/home/james/Desktop/CNN_Malware_Analysis/pickles/binary_stats/bi
narybytesCount_v22_M_output", "rb"))


input_array = np.concatenate((benign_data, malware_data), axis=0)
output_array = np.concatenate((benign_labels, malware_labels), axis=0)

input_df=pd.DataFrame(input_array)
output_df=pd.DataFrame(output_array)

cross_val_num=10

for i in range(cross_val_num):
   X_train, x_test, Y_train, y_test = train_test_split(input_df, output_df, test_size = 0.2,
shuffle=True)  #consider shuffle=True

   feature_size=len(input_array[1])
   first_hidden_layer=16
   second_hidden_layer=4

   classifier=MLPClassifier(solver='adam', alpha=0.001,
hidden_layer_sizes=(first_hidden_layer,second_hidden_layer))

   print ("Training...")
   classifier.fit(X_train,np.ravel(Y_train,order='C'))

   print ("Predicting..")
   y_pred=classifier.predict(x_test)

   y_pred_proba=classifier.predict_proba(x_test)
   y2_test=np.asarray(y_test)
   prob_pred=[]
   index=0
   tp=0
```

```
    tp0B=[]
    tp1M=[]
    tn=0
    tn0B=[]
    tn1M=[]
    fp=0
    fp0B=[]
    fp1M=[]
    fn=0
    fn0B=[]
    fn1M=[]
    for j in y_pred_proba:
        print(j)
        if j[1] > j[0]:
            prob_pred.append(1)
            if y2_test[index] == 1:
                tp+=1
                tp0B.append(j[0])
                tp1M.append(j[1])
            else:
                fp+=1
                fp0B.append(j[0])
                fp1M.append(j[1])
        else:
            prob_pred.append(0)
            if y2_test[index] == 0:
                tn+=1
                tn0B.append(j[0])
                tn1M.append(j[1])
            else:
                fn+=1
                fn0B.append(j[0])
                fn1M.append(j[1])
        index+=1

    print("Accuracy Score")
    print(accuracy_score(y_test, y_pred))
    print("TP\t"+str(tp)+"\tBenign(0): "+str(np.mean(tp0B)) + " +-
"+str(np.std(tp0B))+"\tMalware(1): "+str(np.mean(tp1M))+" +-"+str(np.std(tp1M)))
    print("TN\t"+str(tn)+"\tBenign(0): "+str(np.mean(tn0B)) + " +-
"+str(np.std(tn0B))+"\tMalware(1): "+str(np.mean(tn1M))+" +-"+str(np.std(tn1M)))
    print("FP\t"+str(fp)+"\tBenign(0): "+str(np.mean(fp0B)) + " +-
"+str(np.std(fp0B))+"\tMalware(1): "+str(np.mean(fp1M))+" +-"+str(np.std(fp1M)))
    print("FN\t"+str(fn)+"\tBenign(0): "+str(np.mean(fn0B)) + " +-
"+str(np.std(fn0B))+"\tMalware(1): "+str(np.mean(fn1M))+" +-"+str(np.std(fn1M)))
```

```
    print("Confusion Matrix")
    cm = confusion_matrix(y_test, y_pred)
    print(cm)
    TN=float(cm[0,0])
    FP=float(cm[0,1])
    FN=float(cm[1,0])
    TP=float(cm[1,1])
    Precision=TP/(TP+FP)
    print("Precision: " + str(Precision) +"%")
    Recall=TP/(TP+FN)
    print("Recall: " + str(Recall) + "%")
    F_score=2*((Precision*Recall)/(Precision+Recall))
    print("F-score: " + str(F_score) + "%")

# Model persistence
pickle_clf="/home/james/Desktop/CNN_Malware_Analysis/models/binary_stats_MLP/n
_gram_stats"
pickle.dump(classifier, open(pickle_clf, "wb"))


########### START DYNAMIC-LINK-LIBRARY STRATEGY PART 1 ##########

#!/usr/bin/python

import numpy as np
import pickle
import json

m_dll={}
m_call={}
m_section={}
m_num_sections={}

b_dll={}
b_call={}
b_section={}
b_num_sections={}

analysis_dir='/home/nps/.cuckoo/storage/analyses/'
report_file='/reports/report.json'
start_ben=3252

for analyses_num in range(1,5808):

    report_json=analysis_dir + str(analyses_num) + report_file
```

```
data = json.load(open(report_json))

for imports in data['static']['pe_imports']:
    dll=imports['dll']
    if analyses_num < 3252:
        #Malware
        if dll in m_dll:
            m_dll[dll]+=1
        else:
            m_dll[dll]=1
    else:
        #Benign
        if dll in b_dll:
            b_dll[dll]+=1
        else:
            b_dll[dll]=1

    for name in imports['imports']:
        call=name['name']
        if analyses_num < 3252:
            #Malware
            if call in m_call:
                m_call[call]+=1
            else:
                m_call[call]=1
        else:
            #Benign
            if call in b_call:
                b_call[call]+=1
            else:
                b_call[call]=1

number_of_sections=0
for imports in data['static']['pe_sections']:
    section=imports['name']
    entropy=imports['entropy']
    number_of_sections+=1

    if analyses_num < 3252:
        #Malware
        if section in m_section:
            m_section[section]+=1
        else:
            m_section[section]=1
    else:
            #Benign
            if section in b_section:
                b_section[section]+=1
            else:
                b_section[section]=1

    if analyses_num < 3252:
        #Malware
        if number_of_sections in m_num_sections:
            m_num_sections[number_of_sections]+=1
        else:
            m_num_sections[number_of_sections]=1
    else:
        #Benign
        if number_of_sections in b_num_sections:
            b_num_sections[number_of_sections]+=1
        else:
            b_num_sections[number_of_sections]=1


feat_dict={}
features=[]
output_arr=[]
freq=1300 # Results in Top 10 DLL/Section names
malware=1
benign=0
for key,value in sorted(m_dll.items()):
    if value >= freq:
        feat_dict[key]=1

for key,value in sorted(m_section.items()):
    if value >= freq:
        feat_dict[key]=1

for key,value in sorted(b_dll.items()):
    if value >= freq:
        feat_dict[key]=1

for key,value in sorted(b_section.items()):
    if value >= freq:
        feat_dict[key]=1

for key,value in feat_dict.items():
    features.append(key)
```

```
pickle_in="/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_lib/pe_json"+str
(freq)+"(freq)_input"
np_inputs=np.array(features)
pickle.dump(np_inputs, open(pickle_in, "wb"))


########### DYNAMIC-LINK-LIBRARY STRATEGY PART 2 #################

#!/usr/bin/python

import numpy as np
import pickle
import json

freq=1300 # Results in Top 10 DLL/Section names
feature_set =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_lib/pe_js
on"+str(freq)+"(freq)_input", "rb"))

sample=[]
feature_array=[]
output_array=[]

a_feats={}
a_call={}
a_section={}
m_num_sections={}

b_dll={}
b_call={}
b_section={}
b_num_sections={}

analysis_dir='/home/nps/.cuckoo/storage/analyses/'
report_file='/reports/report.json'
start_ben=3252

for analyses_num in range(1,5808):

    report_json=analysis_dir + str(analyses_num) + report_file
    data = json.load(open(report_json))
    a_feats.clear()
    del sample[:]
    for imports in data['static']['pe_imports']:
        dll=imports['dll']
        a_feats[dll]=1

        for name in imports['imports']:
            call=name['name']
            a_feats[call]=1

    number_of_sections=0
    avg_entropy=0

    for imports in data['static']['pe_sections']:
        section=imports['name']
        entropy=imports['entropy']
        number_of_sections+=1
        avg_entropy+=float(entropy)
        a_feats[section]=1

    for i in range(len(feature_set)):
        if feature_set[i] in a_feats:
            sample.append(1)
        else:
            sample.append(0)

    sample.append(number_of_sections)
    sample.append((avg_entropy/number_of_sections))
    np_sample=np.array(sample)
    feature_array.append(np.copy(np_sample))

    if analyses_num < 3252:
        output_array.append(1)
    else:
        output_array.append(0)

pickle_in="/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_lib/pe_json"+str
(freq)+"(freq)_samples_input"
pickle_out="/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_lib/pe_json"+st
r(freq)+"(freq)_samples_output"
np_inputs=np.array(feature_array)
np_output=np.array(output_array)
pickle.dump(np_inputs, open(pickle_in, "wb"))
pickle.dump(np_output, open(pickle_out, "wb"))
```

```
########### DYNAMIC-LINK-LIBRARY STRATEGY PART 3 #################

#!/usr/bin/python

import numpy as np
import pandas as pd
import seaborn as sns
import pickle
import matplotlib.pyplot as plt

from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn import cross_validation

cross_val_num=1
freq=1300
input_array =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_lib/pe_js
on"+str(freq)+"(freq)_samples_input", "rb"))
output_array =
pickle.load(open("/home/nps/Desktop/CNN_Malware_Analysis/pickles/binary_lib/pe_js
on"+str(freq)+"(freq)_samples_output", "rb"))


input_df=pd.DataFrame(input_array)
output_df=pd.DataFrame(output_array)

for i in range(cross_val_num):
   X_train, x_test, Y_train, y_test = train_test_split(input_df, output_df, test_size = 0.2,
shuffle=True)  #consider shuffle=True

   feature_size=len(input_array[1])
   first_hidden_layer=feature_size
   second_hidden_layer=first_hidden_layer/2
   third_hidden_layer=second_hidden_layer/2

   classifier=MLPClassifier(solver='adam', alpha=0.001,
hidden_layer_sizes=(first_hidden_layer,second_hidden_layer))

   print ("Training...")
   classifier.fit(X_train,np.ravel(Y_train,order='C'))
```

```
print ("Predicting..")
y_pred=classifier.predict(x_test)

y_pred_proba=classifier.predict_proba(x_test)
y2_test=np.asarray(y_test)
prob_pred=[]
index=0
tp=0
tp0B=[]
tp1M=[]
tn=0
tn0B=[]
tn1M=[]
fp=0
fp0B=[]
fp1M=[]
fn=0
fn0B=[]
fn1M=[]
for j in y_pred_proba:
    if j[1] > j[0]:
        prob_pred.append(1)
        if y2_test[index] == 1:
            tp+=1
            tp0B.append(j[0])
            tp1M.append(j[1])
        else:
            fp+=1
            fp0B.append(j[0])
            fp1M.append(j[1])
    else:
        prob_pred.append(0)
        if y2_test[index] == 0:
            tn+=1
            tn0B.append(j[0])
            tn1M.append(j[1])
        else:
            fn+=1
            fn0B.append(j[0])
            fn1M.append(j[1])
    index+=1

print("Accuracy Score")
print(accuracy_score(y_test, y_pred))
```

```
    print("TP\t"+str(tp)+"\tBenign(0): "+str(np.mean(tp0B)) + " +-
"+str(np.std(tp0B))+"\tMalware(1): "+str(np.mean(tp1M))+" +-"+str(np.std(tp1M)))
    print("TN\t"+str(tn)+"\tBenign(0): "+str(np.mean(tn0B)) + " +-
"+str(np.std(tn0B))+"\tMalware(1): "+str(np.mean(tn1M))+" +-"+str(np.std(tn1M)))
    print("FP\t"+str(fp)+"\tBenign(0): "+str(np.mean(fp0B)) + " +-
"+str(np.std(fp0B))+"\tMalware(1): "+str(np.mean(fp1M))+" +-"+str(np.std(fp1M)))
    print("FN\t"+str(fn)+"\tBenign(0): "+str(np.mean(fn0B)) + " +-
"+str(np.std(fn0B))+"\tMalware(1): "+str(np.mean(fn1M))+" +-"+str(np.std(fn1M)))
    print("Confusion Matrix")
    cm = confusion_matrix(y_test, y_pred)
    print(cm)
    TN=float(cm[0,0])
    FP=float(cm[0,1])
    FN=float(cm[1,0])
    TP=float(cm[1,1])
    Precision=TP/(TP+FP)
    print("Precision: " + str(Precision) +"%")
    Recall=TP/(TP+FN)
    print("Recall: " + str(Recall) + "%")
    F_score=2*((Precision*Recall)/(Precision+Recall))
    print("F-score: " + str(F_score) + "%")

# Model persistence
pickle_clf="/home/nps/Desktop/CNN_Malware_Analysis/models/binary_lib_MLP/pe_js
on" + str(freq)
pickle.dump(classifier, open(pickle_clf, "wb"))
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

AV-Test (n.d.). Malware statistics. https://www.av-test.org/en/statistics/malware/

Attaluri, S., Mcghee, S., & Stamp, M. (2008). Profile hidden Markov models and metamorphic virus detection. *Journal in Computer Virology,5*(2), 151–169. doi:10.1007/s11416-008-0105-1

Bui, H. M., Lech, M., Cheng, E., Neville, K., & Burnett, I. S. (2016). Using grayscale images for object recognition with convolutional-recursive neural network. 2016 *IEEE Sixth International Conference on Communications and Electronics* (ICCE). doi:10.1109/cce.2016.7562656

Cuckoo project. (2015, September 02). Analysis packages—Cuckoo. Retrieved 2018, from https://cuckoo.readthedocs.io/en/0.3.1/customization/packages/

Cuckoo project (n.d.) Cuckoo automated malware analysis. Retrieved from https://cuckoosandbox.org/

Cun, Y. L., & Bengio, Y. (n.d.). Word-level training of a handwritten word recognizer based on convolutional neural networks. *Proceedings of the 12th IAPR International Conference on Pattern Recognition (Cat. No.94CH3440-5).* doi:10.1109/icpr.1994.576881

El-Sherei, S. (n.d.) Return oriented programming (ROP FTW)—exploit-db.com. (n.d.). Retrieved from https://www.bing.com/cr?IG=524D343BFE594F36B157B8B5BA6D2126&CID=3D4B8A7AFF83670F3EC0863DFE7E666B&rd=1&h=y6yhMH-J3rCS-CnZwKzpmDP11T_Spomm3gPjArmln48&v=1&r=https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf&p=DevEx.LB.1,5063.1

Fatima, M., & Pasha, M. (2017). Survey of machine learning algorithms for disease diagnostic. *Journal of Intelligent Learning Systems and Applications,09*(01), 1–16. doi:10.4236/jilsa.2017.91001

Gibert, D. (2016). Convolutional neural networks for malware classification. (n.d.). Retrieved from http://www.bing.com/cr?IG=8E1566F46B004D7DA89EAEF26AF2A318&CID=3E77BEAD04526AE72D49B2EA05AF6B8D&rd=1&h=ilFcDqQ7WERSRAG-rWyO1RG-bSJOKEyJwoXeUs4__18&v=1&r=http://www.covert.io/research-papers/deep-learning-security/Convolutional Neural Networks for Malware Classification.pdf&p=DevEx.LB.1,5510.1

Gong, M. (2016). Classifying Windows malware with static analysis. (n.d.). Retrieved from https://www.bing.com/cr?IG=294F2C5136024BF1B5F71A168401AB02&CID=21137EBF1D486EFA0BFB72F81CFF6FA6&rd=1&h=UKHRoBKHhYPq0HafmiYslBJlWEFzUxgDC8yz5WmCSeM&v=1&r=https://courses.csail.mit.edu/6.857/2016/files/5.pdf&p=DevEx.LB.1,5065.1

Hassen, M., Carvalho, M. M., & Chan, P. K. (2017). Malware classification using static analysis based features. *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. doi:10.1109/ssci.2017.8285426

Hollemans, M. (2017, February 22). Matrix multiplication with metal performance shaders. Retrieved 2018, from http://machinethink.net/blog/mps-matrix-multiplication/

IIIT-H Virtual Labs. (n.d.). Tutorial—architecture of multilayer feedforward neural network. Retrieved from http://cse22-iiith.vlabs.ac.in/exp4/index.html

Jahan, R. (2018). Applying Naive Bayes classification technique for classification of improved agricultural land soils. *International Journal for Research in Applied Science and Engineering Technology,6*(5), 189–193. doi:10.22214/ijraset.2018.5030

Kalash, M., Rochan, M., Mohammed, N., Bruce, N. D., Wang, Y., & Iqbal, F. (2018). Malware classification with deep convolutional neural networks. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS).* doi:10.1109/ntms.2018.8328749

Kelleher, J. D., Namee, B. M., & Darcy, A. (2015). *Fundamentals of machine learning for predictive data analytics: Algorithms, worked examples, and case studies.* Massachusetts: The MIT Press.

Kolkiewicz, A. (2010). Stochastic mesh method. *Encyclopedia of Quantitative Finance*. doi:10.1002/9780470061602.eqf13013

Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., & Eckert, C. (2017). Empowering convolutional networks for malware classification and analysis. *2017 International Joint Conference on Neural Networks (IJCNN).* doi:10.1109/ijcnn.2017.7966340

Kolosnjaji, B., Zarras, A., Webster, G., & Eckert, C. (2016). Deep learning for classification of malware system call sequences. *AI 2016: Advances in Artificial Intelligence Lecture Notes in Computer Science*,137-149. doi:10.1007/978-3-319-50127-7_11

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM,60*(6), 84–90. doi:10.1145/3065386

Li, F. (n.d.). CS231n convolutional neural networks for visual recognition. Retrieved 2018, from http://cs231n.github.io/convolutional-networks/

Lopez-Martin, M., Carro, B., Sanchez-Esguevillas, A., & Lloret, J. (2017). Network traffic classifier with convolutional and recurrent neural networks for Internet of things. *IEEE Access,5*, 18042-18050. doi:10.1109/access.2017.2747560

McCarrin, M., Gera, R., Rowe, N., & Allen, B. (2017). Retrieved from https://wiki.nps.edu/display/DEEP/Digital Evaluation and Exploitation (DEEP) Research Group at Naval Postgraduate School.

Microsoft. (2015) Malware Classification Challenge (BIG 2015) : Kaggle. Retrieved from https://www.kaggle.com/c/malware-classification

Muja, M., & Lowe, D. G. (2014). Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence,36*(11), 2227–2240. doi:10.1109/tpami.2014.2321376

Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware images. *Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec 11*. doi:10.1145/2016904.2016908

Osmanbeyoglu, H., & Ganapathiraju, M. K. (2011). N-gram analysis of 970 microbial organisms reveals presence of biological language models. *BMC Bioinformatics,12*(1), 12. doi:10.1186/1471-2105-12-12

Pfoh, J., Schneider, C., & Eckert, C. (2013). Leveraging string kernels for malware detection. *Network and System Security Lecture Notes in Computer Science,*206-219. doi:10.1007/978-3-642-38631-2_16

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B, Nichols, C. (2017, October 25). Malware detection by eating a whole EXE. Retrieved from https://arxiv.org/abs/1710.09435

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review,65*(6), 386–408. doi:10.1037/h0042519

Spark, C. (2017, March 20). Deep learning for complete beginners: convolutional neural networks with Keras. Retrieved 2018, from https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

Stallman, R. (1983). Objdump(1)—Linux main page. Retrieved 2018, from https://linux.die.net/man/1/objdump

Symantec Corporation (2018, April). 2018 Internet security threat report. Retrieved from https://www.symantec.com/security-center/threat-report

TensorFlow (2018, August 8). Build a convolutional neural network using estimators | TensorFlow. Retrieved from https://www.tensorflow.org/tutorials/estimators/cnn

VirusShare (n.d.). VirusShare. Retrieved from http://virusshare.com/

VMWare (n.d.). VMware—Official Site. Retrieved 2018, from https://www.vmware.com/

Zak, R., Raff, E., & Nicholas, C. (2017). What can n-grams learn for malware detection? *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. doi:10.1109/malware.2017.8323963

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California